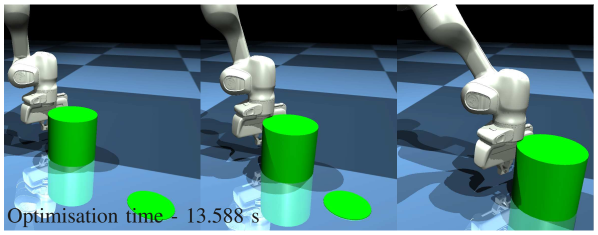
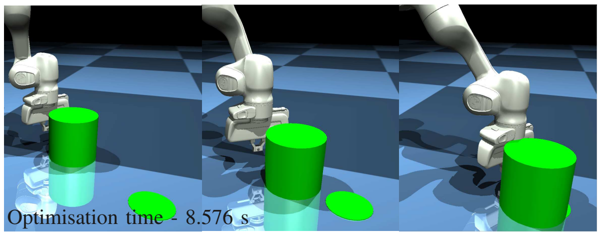
Adaptive approximation of dynamics gradients via interpolation to speed up trajectory optimisation

David Russell, Rafael Papallas and Mehmet Dogar

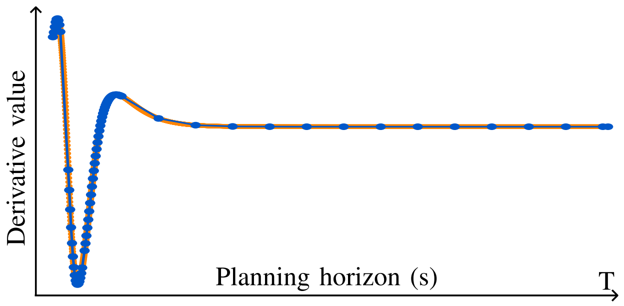
***Abstract*— Trajectory optimisation methods for robotic motion planning often require the use of first order derivatives of the dynamics of the system with respect to the states and controls of the system. Particularly when multi-contact dynamics are present, these derivatives are often numerically approximated by a method such as finite-differencing. Finite-differencing whilst using an expensive physics simulator is usually the bottleneck in these trajectory optimisation algorithms. Since these dynamics derivatives do not change substantially over certain time intervals, we propose that trajectory optimisers can compute the dynamics derivatives less often and then interpolate approximations to the derivatives in between calculated derivatives, gaining a significant speed up for overall optimisation time with no observable degradation in the generated behaviour. We investigate different methods of interpolating approximations as well as propose an adaptive method to detect when to compute the derivatives with finite-differencing. We find a speed-up of planning times on average by 60% in a contact-based manipulation task.**



(a) Object pushing trajectory generated by the **baseline-iLQR** method



(b) Trajectory generated by the **adaptive interpolated-iLQR** method



(c) Adaptive linear interpolation of a dynamics derivative

Fig. 1: (a) and (b) show two finalised trajectories, using the baseline-iLQR and our adaptive interpolated-iLQR. The times on the bottom-left show how long the optimisations took. (c) shows the values of the derivatives of a particular state variable with respect to another during optimisation. Orange points are computed using finite-differencing at every time-step. Our method performs fewer finite-differencing operations (blue points) and then interpolates between the blue points.

I. INTRODUCTION

In this work, we propose and compare methods to speed up trajectory optimisation for robot motion generation. For example, in Fig. 1-(a), a robot uses a trajectory optimisation method (particularly the *iterative linear quadratic regulator*, iLQR [1], [2]), to push an object (green cylinder) to a goal location (green circle). The time it takes to optimise such a trajectory is also shown on the top-right. We propose different methods of interpolating the system dynamics gradients to perform the same iLQR optimisation but computationally cheaper. Fig. 1-(b) shows the motion generated using our method, which is similar to (a), while the optimisation takes significantly less time. While we use iLQR in this work, our proposed method can work with any trajectory optimisation algorithm that utilises dynamics gradients.

Trajectory optimisation methods have been developed and used extensively recently [3]–[9]. iLQR is one such widely used algorithm [10]–[14] which has also been applied to problems that involve contact [12], [15], thanks to smooth contact models such as those implemented in MuJoCo [16] and Drake [17]. These optimisation techniques use gradient information on the system's dynamics and a cost function to calculate the optimal trajectory.

While effective, trajectory optimisation methods can be computationally expensive, and therefore the optimisation time can be prohibitively long, particularly for problems involving robots with high degrees of freedom (DoF) and when the robot makes contact with objects or the environment. For example, Kitaev et al. [15] apply iLQR to the problem of manipulation in clutter, where the robot pushes through multiple objects on a shelf. They report optimisation times on the order of tens of seconds and sometimes over a minute, depending on the level of clutter. Mordatch et al. [18] address contact-based in-hand manipulation problem using an optimisation approach, and report 2-6 minutes of optimisation time. Speeding up

This research has received funding from the UK Engineering and Physical Sciences Research Council under the grants EP/V052659/1 and 2596473.

Authors are with the School of Computing, University of Leeds, UK. {el16dmcr, r.papallas, m.r.dogar}@leeds.ac.uk

optimisation is therefore important in such settings, not only because this is a problem for planning, but also because contact-based manipulation in the real world often requires re-running the optimiser during execution (e.g. using model-predictive-control [19], [20] or online re-planning [21], [22]).

One reason why trajectory optimisation can take a long time is the evaluation of the gradients of the system dynamics and/or the cost function, with respect to the state variables and controls. If analytical expressions of gradients are available, they can be directly used. However, for multi-DoF robot systems, coming up with analytical expressions can be difficult or impossible, especially when dynamic (i.e. non-quasi-static) contact is made with (possibly multiple) objects, since in such cases the system dynamics itself is formulated as a separate optimisation/complementarity problem [23], [24].

When the analytic gradients are not available, a common method [15], [18] is to estimate the gradients numerically using finite-differencing [25], which involves making small changes to states of the system and evaluating the effect this has on the system dynamics. Using finite-differencing for trajectory optimisation requires performing such an evaluation a large number of times at every time-step along the trajectory. For robotic manipulation, evaluating the system dynamics usually corresponds to integrating a physics engine, which is a computationally expensive operation [26]. Therefore decreasing the number of finite-differencing operations during trajectory optimisation has the potential to significantly reduce the optimisation time. Here, we investigate such an approach.

Our key insight is that gradient/derivative values over a trajectory generally show regularity making them amenable to interpolation. For example, in Fig. 1-(c), we show the value of an individual derivative computed via finite-differencing at every time-step (orange dots) over a trajectory. Instead of computing every derivative, we can simply compute a selection of them and interpolate approximations in between.

While we use finite-differencing in this work, the interpolation approach we propose can also be used when alternative methods of calculating system dynamics gradients are used, such as *differentiable simulation* [17], [27], [28]. For example, [14] use a differentiable physics simulator to plan contact-based manipulation tasks. Their optimisation takes a prohibitively long time which they claim is due to the automatic differentiation scheme. Therefore, automatic differentiation can also benefit from our method.

We propose and compare different methods to perform such an interpolation. We investigate two questions. The first question is about how to determine, during optimisation, the key time-steps at which to perform finite-differencing. We propose two methods: (i) Using fixed-size intervals and interpolating within the intervals. This method does help reduce the optimisation time while enabling the optimisation to converge. However, we notice that, depending on the task, the system dynamics, and the variables in question, the sizes of the intervals in which the values show regularity change. Therefore, we also propose and evaluate a second method; (ii) Adaptively determining the intervals. For example, in Fig. 1-(c) the blue points show the key time-steps (where this method performs finite-differencing). As can be seen, the distance between the blue points get smaller in areas of the plot where there is high-curvature and is therefore more difficult to interpolate. Whereas in areas where the values show more regularity, the intervals between the blue points get larger, and therefore we need fewer finite-differencing operations. We propose a method to identify such key time-steps.

The second question is about how to perform the interpolation within the intervals, given the computed values at the key time-steps. We compare three different methods: (i) linear interpolation, (ii) quadratic interpolation, and (iii) neural network interpolation. Particularly, Cheng et al. [10] also used a neural network to estimate the first order dynamics gradients during iLQR. They trained the network to predict the gradient based on the state vector as input. Differently, we take an interpolation approach, and therefore we also provide the neural network the two gradient matrices to interpolate in-between.

We integrate the dynamics interpolation methods described above with iLQR optimisation, and compare it with a baseline iLQR implementation which performs the finite-differencing at every time-step. We investigate whether we can achieve similar performance in terms of cost minimisation, while reducing the optimisation time. We perform this comparison for a variety of tasks: (i) a pendulum control problem, (ii) reaching a desired configuration of a 7-DOF robotic arm whilst minimizing torque controls and joint velocities, and (iii) pushing an object using a 7-DOF robotic arm to a desired location (Fig. 1). In all tasks, we observe that it is possible to achieve cost minimisation equivalent to the baseline iLQR, while significantly reducing the optimisation time. We provide our source code[1] as well as the data associated with this paper.

In Sec. II we provide a brief overview of the iLQR algorithm and finite-differencing; in Sec. III we discuss the different interpolation methods and how they are integrated with iLQR. Finally, we present and discuss our results in Sec. IV.

II. BACKGROUND

Consider a discrete-time dynamics problem where the next system state is a function of the previous state and control:

$$\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{u}(t)) \tag{1}$$

where our state vector is typically the joints of the robot arm as well as their velocities and the control vector is the torques applied to the joints. In the object-pushing task, the state vector also includes the object's pose and velocity.

We have some cost function $l(\mathbf{x}(t), \mathbf{u}(t))$ and the total running cost from the start of the trajectory to the end is:

$$J_0(\mathbf{x}(0), \mathbf{U}) = l_f(\mathbf{x}(T)) + \sum_{t=0}^{T-1} l(\mathbf{x}(t), \mathbf{u}(t)) \tag{2}$$

where l_f is a terminal cost function, T is the horizon length and $\mathbf{U}$ is a sequence of control vectors over the trajectory. The value function at any time-step is the optimal cost to go from that time-step and state.

$$\mathbf{V}(\mathbf{x}(t), t) = \min_{\mathbf{U}} [J_t(\mathbf{x}(t), \mathbf{U})] \tag{3}$$

[1] github.com/DMackRus/Dynamic_Interpolation_iLQR/tree/ICRA_2023

The overall minimisation problem that we are trying to solve is, given an initial state $\mathbf{x}(0)$, find the optimal controls that minimise the total running cost of the trajectory.

$$\mathbf{U}^* = \min_{\mathbf{U}}[J_0(\mathbf{x}(0), \mathbf{U})] \tag{4}$$

A. Iterative Linear Quadratic Regulator

The iLQR algorithm [1], [2] solves the optimisation problem from Eq. 4 by calculating the first order dynamics f_x and f_u, as well as the first and second order cost derivatives l_x, l_u, l_{xx} and l_{uu} over a trajectory, and calculating an optimal closed-loop control policy that minimises the total cost of the trajectory. The high-level overview of this algorithm is in Alg. 1.

The algorithm uses these derivatives to perform a backwards pass in which it recursively computes feedback terms k and K from the end of the trajectory to the beginning. After this, a forwards pass is carried out using the feedback terms to calculate a new set of controls which will lower the cost of the trajectory. Finally, the algorithm checks for convergence and either repeats this optimisation process or returns the optimised trajectory. iLQR details are in [1], [2].

B. Computing dynamics gradients with finite-differencing

The first order dynamics derivatives f_x, f_u are matrices with size relative to the size of the state and control vector. A state vector of size $m \times 1$ and a control vector of $q \times 1$ will result in matrices f_x, f_u of size $m \times m$ and $m \times q$ respectively. We will use square brackets to refer to an individual element within these matrices; so $f_x[i, j]$ will represent an element at the i^{th} row and j^{th} column. $f_x[:, j]$ will refer to an entire column of this matrix at the j^{th} position.

Computation of the f_x, f_u matrices works column by column with finite-differencing. Given a specific state and control vector $\mathbf{x}(t), \mathbf{u}(t)$, a small perturbation ϵ is added/subtracted to an individual element of the state vector and it is observed how this affects the overall state, giving one column of gradient values. The process of calculating an f_x matrix can be seen in Alg. 2 and is similar for the f_u matrix but the control vectors are perturbed instead. We will refer to this process as performing one **dynamics gradient evaluation**.

Algorithm 1 iLQR

```
Require: Prediction horizon $T$
    Maximum number of iLQR iterations $i_{max}$
    Starting state $\mathbf{x}(0)$ and initialised control sequence $\mathbf{U}$
1: for $iteration = 1, 2, \ldots i_{max}$ do
2:     Step 1: (Get Derivatives)
3:     for $t = 0, 1, \ldots T$ do
4:         Calculate $f_x(t)$ $f_u(t)$ $l_x(t)$ $l_{xx}(t)$ $l_u(t)$ $l_{uu}(t)$
5:     Step 2: (Backwards Pass)
6:     Calculate $k$ and $K$ using derivatives
7:     Step 3: (Forwards Pass)
8:     Update $\mathbf{U}$ using $k$ and $K$
9:     Step 4: (Convergence Check)
   return U
```

Algorithm 2 FD_DynamicsGradientEval()

```
Require: State vector $\mathbf{x}(t)$; Control vector $\mathbf{u}(t)$
1: for $i = 0, 1, \ldots, (m-1)$ do
2:     $\mathbf{x}_{inc}(t) = \mathbf{x}(t) + \epsilon$
3:     $\mathbf{x}_{dec}(t) = \mathbf{x}(t) - \epsilon$
4:     $f^*_x(t)[:, i] = \frac{f(\mathbf{x}_{inc}(t), \mathbf{u}(t)) - f(\mathbf{x}_{dec}(t), \mathbf{u}(t))}{2\epsilon}$
```

Algorithm 3 *Interpolated-iLQR*: Alg. 1 with Step 1 (lines 2-4) implemented as below

```
1: FD_timeSteps ← GenerateFDTimesteps()
2: for $i = 0, 1, \ldots$ len(FD_timeSteps) do
3:     $f^*_x$(FD_timeSteps[i]) = FD_DynamicsGradientEval()
4: $\tilde{f_x}$ = Interpolate($f^*_x$)
```

To perform one dynamics gradient evaluation, $2m + 2q$ evaluations of f are required. The function f is the process of integrating the system dynamics with time. When performing optimisation with a physics simulator such as MuJoCo [16], integrating the system dynamics with time (referred to as **stepping** the simulator) is computationally expensive.

C. Baseline-iLQR

The baseline method of iLQR referred to in the rest of this paper will be where in Step 1 (lines 2-4) of Alg. 1 **all** of the matrices f_x and f_u along the entire trajectory are computed by finite-differencing as shown in Alg. 2. We will refer to these types of gradient matrices as the **true** matrices and they will be denoted with an asterisk symbol ($f^*_x(t)[i, j]$). Therefore, each Baseline-iLQR iteration, for a trajectory of length T, requires $2mT + 2qT$ steps of the simulator.

III. METHOD

Our method involves performing these expensive finite-differencing computations only at certain time-steps along the trajectory (which we will call **finite-difference time-steps** or **FD_timeSteps** for short) and then interpolating approximations to the dynamics gradient at time-steps in between. These approximations will be denoted as $\tilde{f_x}(t)[i, j]$. The number of time-steps between two FD_timeSteps will be referred to as the **interval**.

Alg. 3 shows the high-level overview of our method. In line 1 we generate the list of FD_timeSteps, then in lines 2 and 3, we iterate over these time-steps and numerically calculate the dynamics gradients. Finally, in line 4, we use an interpolation algorithm to calculate approximations for the remaining time-steps where finite-differencing was not performed. In this section, any algorithm that performs an operation on a matrix uses element-wise operation. We only discuss f_x matrices, but the method is identical for the f_u matrices.

In the next sections, we propose multiple different ways to implement Alg. 3, using different methods for the Interpolate() and GenerateFDTimesteps() function. Particularly, we investigate two different ways to implement GenerateFDTimesteps():

fixed-interval methods (described in detail in Sec. III-A) and *adaptive-interval methods* (described in Sec. III-B).

A. Fixed-Interval Methods

The fixed-interval methods work by equally spacing out the dynamics gradient evaluations along the trajectory, i.e. the FD_timeSteps are simply chosen with a fixed interval between them. Therefore, for an interval size of n, this method implements the GenerateFDTimesteps() function in Alg. 3 as:

$$[0, n, 2n, \dots, T] \leftarrow \text{GenerateFDTimesteps()}$$

and, therefore, at each iteration of iLQR, we would only calculate (using finite-differencing as in Alg. 2) the f_x^* matrices: $f_x^*(0), f_x^*(n), f_x^*(2n), \dots, f_x^*(T)$.

Using these values at the FD_timeSteps, we then compare three different methods to interpolate $\tilde{f}_x(t)$ at other time-steps, i.e. three different ways to implement Interpolate() in Alg. 3.

Linear Interpolation - We interpolate a line between two computed derivatives. This can be seen in Alg. 4.

Algorithm 4 Interpolate() - Linear

```
Require: Finite-differencing time-steps, FD_timeSteps
1: for $i = 0, 1, \dots,$ len(FD_timeSteps)$-1$ do
2:     startGradient $= f_x^*$(FD_timeSteps[$i$])
3:     endGradient $= f_x^*$(FD_timeSteps[$i+1$])
4:     distance = FD_timeSteps[$i+1$] - FD_timeSteps[$i$]
5:     difference = endGradient - startGradient
6:     for $j = 1, 2 \dots,$ distance$-1$ do
7:         $\tilde{f}_x$(FD_timeSteps[$i$]$+j$) = $f_x^*$(FD_timeSteps[$i$])$+$ ($\frac{j}{\text{distance}} *$ difference)
   return $\tilde{f}_x$
```

Quadratic Interpolation - We fit a quadratic curve between three derivatives and interpolate along it (Alg. 5).

Algorithm 5 Interpolate() - Quadratic

```
Require: Finite-differencing time-steps, FD_timeSteps
1: for $i = 0, 1, \dots,$ len(FD_timeSteps)$-2$ do
2:     startGradient $= f_x^*$(FD_timeSteps[$i$])
3:     midGradient $= f_x^*$(FD_timeSteps[$i+1$])
4:     endGradient $= f_x^*$(FD_timeSteps[$i+2$])
5:     distance = FD_timeSteps[$i+1$]$-$FD_timeSteps[$i$]
6:     $a, b, c$ = fitQuadratic(startGradient, midGradient, endGradient)
7:     for $j = 1, 2 \dots,$ distance$-1$ do
8:         $\tilde{f}_x$(FD_timeSteps[$i$]$+j$) $= aj^2 + bj + c$
   return $\tilde{f}_x$
```

Neural Network Interpolation - NN-Interpolation uses a trained neural network to output desired dynamic gradient matrices. We trained a separate neural network for each task as the number of inputs and outputs were dependant on the size of the f_x matrices.

The inputs to the neural network were the two calculated matrices that are being interpolated in-between (f_x^*(FD_timeSteps[i]), f_x^*(FD_timeSteps[i+1])), the current state vector $\mathbf{x}(t)$ and a scale parameter $s \in \{0, 1\}$ which represents how far along the interval (between FD_timeSteps[i] and FD_timeSteps[i+1]) the interpolation time, t, is. The algorithm for using NN-Interpolation is in Alg. 6.

We created training data by generating 1000 example trajectories with an optimisation horizon of 12 seconds for each task where **all** of the f_x^* matrices were computed via finite-differencing. During training, we systematically picked gradient matrices at equal intervals and then changed the scale and state vector parameters to coincide with a gradient matrix that was in between the two gradient matrices that were being interpolated. An example training data row can be seen in the table below.

Inputs				**Outputs**
$f_x^*(0)$	$f_x^*(20)$	$\mathbf{x}(10)$	0.5	$f_x^*(10)$

Algorithm 6 Interpolate() - NN

```
Require: Finite-differencing time-steps, FD_timeSteps
1: for $i = 0, 1, \dots,$ len(FD_timeSteps) $-1$ do
2:     startGradient $= f_x^*$(FD_timeSteps[$i$])
3:     endGradient $= f_x^*$(FD_timeSteps[$i+1$])
4:     distance = FD_timeSteps[$i+1$]$-$FD_timeSteps[$i$]
5:     for $j = 1, 2 \dots,$ distance$-1$ do
6:         $\tilde{f}_x$(FD_timeSteps[$i$] $+$ $j$) = queryNetwork(startGradient, endGradient, $\mathbf{x}$(FD_timeSteps[$i$]$+j$), $\frac{j}{\text{distance}}$)
   return $\tilde{f}_x$
```

B. Adaptive-Interval Methods

Here, we describe an alternative method to implement the GenerateFDTimesteps() function. The *adaptive-interval* method computes "key points" where the gradients change significantly to reduce the number of dynamic gradient evaluations over long periods where the dynamics are fairly linear.

We illustrate the difference between the fixed-interval method (Sec. III-A) and the adaptive-interval method, in Fig. 2. The green dots show the finite-differencing time-steps as suggested by the fixed-interval method for a particular interval size n. The blue dots illustrate the adaptive method which accurately models the underlying curve with minimal keypoints. We illustrate here that an adaptive approach to interpolation can model the underlying **true** dynamics function more accurately and with fewer points.

We propose that by analysing the dynamics of the system over a trajectory, we can compute where to place keypoints, such that we can space them out further in periods of linearity and keep them closer together when the derivatives are changing rapidly. We achieve this by analysing the jerk (the third time derivative) of all the DoFs in the system over a trajectory and monitor when the jerk exceeds a certain threshold. While there may be other (and possibly better) methods to determine these keypoints, we found that the jerk-based analysis performed well.

Our algorithm to generate "keypoints" where finite-differencing should occur is shown in Alg. 7. As well as

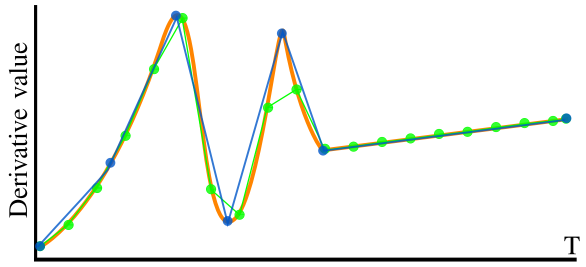


Fig. 2: Illustrated example of the derivative value of an individual element ($f_x[i, j]$) over a full trajectory. Orange is the true gradient ($f_x^*[i, j]$), and blue and green are adaptive and fixed-interval dynamics gradient evaluations respectively.

checking for when the jerk of any DoF exceeds a certain threshold, we also specify a minimum and maximum interval size that should be followed. After the finite-differencing time-steps are determined adaptively, the interpolation is performed using linear interpolation from Alg. 4.

IV. EXPERIMENTS AND RESULTS

We present two sets of results. First, we tested the four different interpolation methods on pre-computed trajectories where all of the f_x matrices were calculated via finite-differencing and we measured the accuracy of these methods with different interval sizes. We choose the better performing methods, with which we then conduct our second set of experiments, where we combine them with iLQR.

We implemented our methods in C++ (available in our github repo) and used MuJoCo [16] as the physics simulator, with an internal integration step-size of 0.004 s. Our trajectories had a fixed horizon time of 12 s. We used an Intel i7-6700 CPU that has 8 cores and 32GB of RAM.

A. *Accuracy of interpolation methods*

We generated 10 trajectories for the reaching task and tested different interpolation methods by measuring the accuracy of the interpolated matrices ($\tilde{f}_x$) against the **true** matrices (f_x^*). We tested intervals of (2, 5, 10, 20, 50, 100 and 200) for the fixed-interval methods. For the adaptive method we tested different parameters for (minN, maxN, jerkThresh). These parameters were:{{2, 5, 0.01}, {5, 50, 0.0001}, {20, 50, 0.0001}, {50, 100, 0.01}, {2, 50, 0.0001}, {2, 100, 0.0001}, {10, 50, 0.001}}. The tested neural network had two hidden

Algorithm 7 GenerateFDTimesteps() - Adaptive

```
Require: minN, maxN, jerkThresh
 1: counter = 0
 2: for $t = 0, 1, \dots, T$ do
 3:     counter++
 4:     if counter > minN then
 5:         if jerk > jerkThresh then
 6:             counter = 0
 7:             FD_timeSteps.append($t$)
 8:     else if counter > maxN then
 9:         counter = 0
10:         FD_timeSteps.append($t$)
11: return FD_timeSteps
```

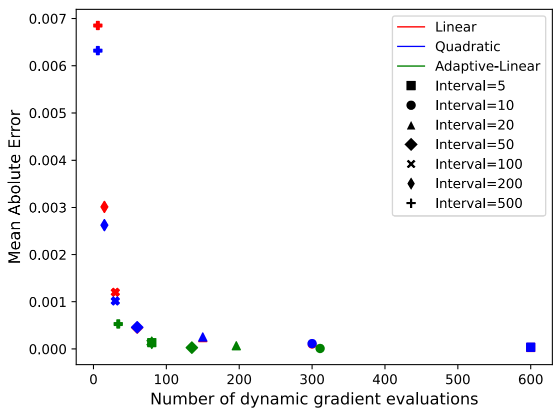


Fig. 3: Mean absolute error for the reaching task for three interpolation methods with different parameters.

layers with 4000 neurons using the ReLU activation function. We used the mean absolute error (MAE) to measure accuracy:

$$\text{MAE} = \sum_{t=0}^{T}\Big(\sum_{i=0,j=0}^{n,n} abs(f_x^*(t)[i,j] - \tilde{f}_x(t)[i,j]/n^2)/T\Big) \tag{5}$$

Fig. 3 shows the average MAE over the ten sample trajectories for the reaching task. The NN-interpolation errors, much larger compared to the other methods, over the specified interval sizes were $(0.04, 0.05, 0.05, 0.05, 0.05, 0.05, 0.06)$, and are not shown in Fig. 3 since they scale the plot poorly. In the pendulum control example NN had the highest accuracy, but as the number of inputs and outputs increased, the performance reduced substantially. It is always possible to increase the number of neurons or hidden layers of a NN and hope for better performance, but since linear interpolation performed well and was computationally cheap, we did not try more expensive NN architectures.

In Fig. 3, the closer to the bottom-left corner a method can get, the better it is. As can be seen, the three methods (linear, quadratic and adaptive-linear) all performed similarly well over different interval sizes, with the adaptive method performing slightly better. The quadratic method was more computationally expensive with no added benefit so we decided to combine the fixed-interval-linear and adaptive-linear interpolation methods with iLQR.

B. *iLQR performance*

To evaluate the performance of our methods when combined with iLQR, we measured the *optimisation time* for iLQR to converge to a solution, with a cap of 15 iLQR iterations. We also measured the *cost reduction* by comparing the optimised trajectory's cost with the initial trajectory's cost:

$$\text{Cost reduction} = 1 - \frac{\text{Final cost}}{\text{Initial cost}} \tag{6}$$

We ran experiments for the three different tasks specified in Sec. I. For each task, we generated 50 random scenes with different starting and desired states. We present the results in Fig. 4. In Fig. 4, the green triangles show the mean values,

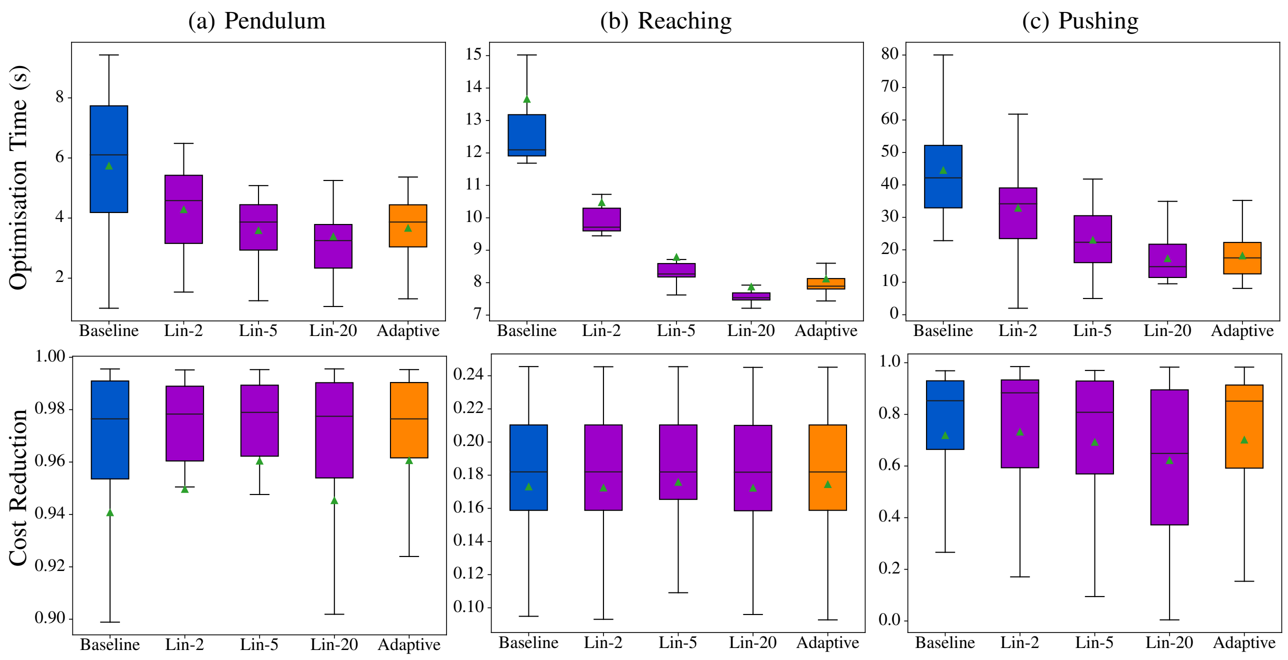


Fig. 4: The top row shows the optimisation time of the iLQR algorithm and the bottom row shows the cost reduction for different interpolation methods for the three different tasks. Blue is the baseline-iLQR method, purple are fixed-interval methods and orange is for the adaptive method. Outliers are not shown.

the line in each box shows the median, and each box shows the interquartile range. We present baseline-iLQR (blue box), fixed-interval linear interpolation (purple box) for different interval sizes, as well as adaptive linear interpolation (orange box). Adaptive linear interpolation was tested with a $minN = 5$, $maxN = 50$, and the $jerkThreshold$ was 0.0001 for robot arm joints and 0.0005 for the pushed-cylinder.

The mean and standard deviation values are also presented in Table I. Please see the video on our github repository for example optimised trajectories for different tasks.

For the pendulum and reaching tasks, there is a noticeable decrease in optimisation time as the interval size gets larger. There is only a minor decrease in the quality of the final solution (i.e. cost reduction) for the pendulum task whereas the reaching tasks final solutions are fairly consistent. The adaptive method, while decreasing optimisation time noticeably, also performs similarly in terms of cost reduction.

The pushing task gives the most interesting result and is the most difficult one out of these three. Calculating the dynamics derivatives for a contact-based task is more expensive than in a non-contact based task, because MuJoCo solves an optimisation problem to calculate contact forces between objects, which becomes more expensive with more bodies.

There is a significant time saving in the overall optimisation time from an interval of 1 time-step to 20 time-steps by 62.6%, however the quality of the solution does start to decrease in the pushing task after an interval of about 10 time-steps. The adaptive method performs very well however. It generally converges to an optimised solution in a similar amount of time as the fixed 20 time-step method, whilst maintaining a cost reduction similar to the fixed 2 or 5 time-step methods. The mean interval size of the adaptive method was 15.6 time-steps, with variance of 304.6, showing that the adaptive method could exploit periods of the trajectory where the dynamics were fairly linear by changing the interval size significantly.

TABLE I: iLQR performance data from Fig 4. Values shown are mean values with standard deviation in parentheses.

Optimisation time			
Interpolation Method	Pendulum	Reaching	Pushing
Baseline	5.89(2.06)	13.83(4.62)	45.74(15.33)
Lin-2	4.36(1.47)	10.59(2.57)	33.34(10.09)
Lin-5	3.58(1.02)	8.82(1.77)	23.32(8.10)
Lin-20	3.07(0.95)	7.60(0.88)	17.09(7.03)
Adaptive	3.67(1.02)	8.12(1.34)	18.18(7.21)
Cost reduction			
Interpolation Method	Pendulum	Reaching	Pushing
Baseline	0.94(0.01)	0.17(0.05)	0.72(0.28)
Lin-2	0.95(0.09)	0.17(0.06)	0.73(0.29)
Lin-5	0.96(0.07)	0.18(0.05)	0.69(0.30)
Lin-20	0.95(0.09)	0.17(0.06)	0.62(0.30)
Adaptive	0.96(0.07)	0.17(0.05)	0.70(0.29)

V. FUTURE WORK

From this work we have concluded that the method of determining the key points is more important than the method of interpolation. In the future, we wish to enhance our adaptive method of determining key points so that an accurate approximation of the dynamics gradients can be calculated with minimal finite-differencing calculations.

We also wish to extend our tasks to also include scenes with medium levels of clutter where calculating dynamics gradients for a large number of objects gets prohibitively expensive.

References

[1] W. Li and E. Todorov, “Iterative linear quadratic regulator design for nonlinear biological movement systems,” in *ICINCO (1)*. Citeseer, 2004, Conference Proceedings, pp. 222–229.

[2] Y. Tassa, T. Erez, and E. Todorov, “Synthesis and stabilization of complex behaviors through online trajectory optimization,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, Conference Proceedings, pp. 4906–4913.

[3] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, “Chomp: Gradient optimization techniques for efficient motion planning,” in *2009 IEEE International Conference on Robotics and Automation*. IEEE, 2009, Conference Proceedings, pp. 489–494.

[4] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, “Stomp: Stochastic trajectory optimization for motion planning,” in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, Conference Proceedings, pp. 4569–4574.

[5] M. Posa, C. Cantu, and R. Tedrake, “A direct method for trajectory optimization of rigid bodies through contact,” *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 69–81, 2014.

[6] S. Jin, D. Romeres, A. Ragunathan, D. K. Jha, and M. Tomizuka, “Trajectory optimization for manipulation of deformable objects: Assembly of belt drive units,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 10 002–10 008.

[7] I. Chatzinikolaidis and Z. Li, “Trajectory optimization of contact-rich motions using implicit differential dynamic programming,” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2626–2633, 2021.

[8] D. Driess, J.-S. Ha, M. Toussaint, and R. Tedrake, “Learning models as functionals of signed-distance fields for manipulation planning,” in *Conference on Robot Learning*. PMLR, 2022, pp. 245–255.

[9] A. Aydinoglu and M. Posa, “Real-time multi-contact model predictive control via admm,” in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 3414–3421.

[10] Z. Cheng, J. Ma, X. Zhang, F. L. Lewis, and T. H. Lee, “Neural network ilqr: A reinforcement learning architecture for trajectory optimization,” *arXiv preprint arXiv:2011.10737*, 2020.

[11] T. Zong, L. Sun, and Y. Liu, “Reinforced ilqr: A sample-efficient robot locomotion learning,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, Conference Proceedings, pp. 5906–5913.

[12] V. Kurtz and H. Lin, “Contact-implicit trajectory optimization with hydroelastic contact and ilqr,” *arXiv preprint arXiv:2202.13986*, 2022.

[13] N. J. Kong, G. Council, and A. M. Johnson, “ilqr for piecewise-smooth hybrid dynamical systems,” in *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE, 2021, Conference Proceedings, pp. 5374–5381.

[14] T. Dinev, C. Mastalli, V. Ivan, S. Tonneau, and S. Vijayakumar, “Differentiable optimal control via differential dynamic programming,” 2022. [Online]. Available: https://arxiv.org/abs/2209.01117

[15] N. Kitaev, I. Mordatch, S. Patil, and P. Abbeel, “Physics-based trajectory optimization for grasping in cluttered environments,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, Conference Proceedings, pp. 3102–3109.

[16] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, Conference Proceedings, pp. 5026–5033.

[17] R. Tedrake and the Drake Development Team, “Drake: Model-based design and verification for robotics,” 2019. [Online]. Available: https://drake.mit.edu

[18] I. Mordatch, Z. Popović, and E. Todorov, “Contact-invariant optimization for hand manipulation,” in *Proceedings of the ACM SIGGRAPH/Eurographics symposium on computer animation*, 2012, pp. 137–144.

[19] F. R. Hogan and A. Rodriguez, “Feedback control of the pusher-slider system: A story of hybrid and underactuated contact dynamics,” *arXiv preprint arXiv:1611.08268*, 2016.

[20] G. Williams, A. Aldrich, and E. A. Theodorou, “Model predictive path integral control: From theory to parallel computation,” *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 2, pp. 344–357, 2017.

[21] W. C. Agboh and M. R. Dogar, “Real-time online re-planning for grasping under clutter and uncertainty,” in *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*. IEEE, 2018, pp. 1–8.

[22] R. Papallas, A. G. Cohn, and M. R. Dogar, “Online replanning with human-in-the-loop for non-prehensile manipulation in clutter—a trajectory optimization based approach,” *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 5377–5384, 2020.

[23] D. E. Stewart and J. C. Trinkle, “An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction,” *International Journal for Numerical Methods in Engineering*, vol. 39, no. 15, pp. 2673–2691, 1996.

[24] M. Anitescu and F. A. Potra, “Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems,” *Nonlinear Dynamics*, vol. 14, no. 3, pp. 231–247, 1997.

[25] J. T. Betts, *Practical methods for optimal control and estimation using nonlinear programming*. SIAM, 2010.

[26] T. Erez, Y. Tassa, and E. Todorov, “Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx,” in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

[27] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, “NeuralSim: Augmenting differentiable simulators with neural networks,” in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2021. [Online]. Available: https://github.com/google-research/tiny-differentiable-simulator

[28] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, “Difftaichi: Differentiable programming for physical simulation,” *arXiv preprint arXiv:1910.00935*, 2019.